

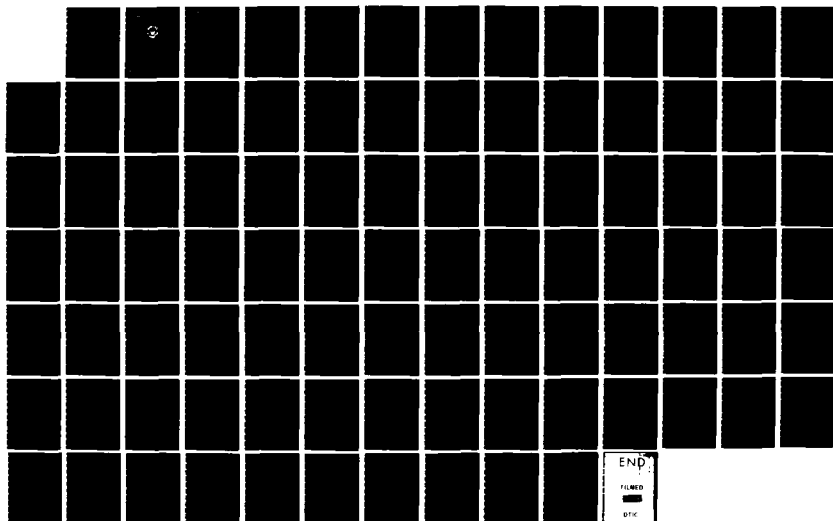
ND-A137 413

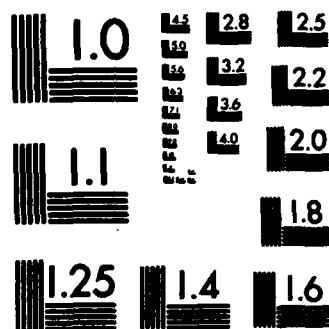
BENCHMARKING RELATIONAL DATABASE MACHINES' CAPABILITIES 1/1
IN SUPPORTING THE (U) NAVAL POSTGRADUATE SCHOOL
MONTEREY CA C M RYDER SEP 83

UNCLASSIFIED

F/G 5/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California



DTIC
FEB 3 1984

THESIS

**BENCHMARKING RELATIONAL DATABASE MACHINES'
CAPABILITIES IN SUPPORTING THE DATABASE ADMINISTRATORS'
FUNCTIONS AND RESPONSIBILITIES**

by

Curtis M. Ryder

September 1983

Thesis Advisor:

D. K. Hsiao

Approved for public release, distribution unlimited

84 02 03 054

AD A 137413

DTIC FILE COPY

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Benchmarking Relational Database Machines' Capabilities in Supporting the Database Administrators' Functions and Responsibilities		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis September 1983
7. AUTHOR(s) Curtis M. Ryder		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
12. REPORT DATE September 1983		13. NUMBER OF PAGES 91
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Database Administrator (DBA), Relational Query Language (RQL), Benchmarking a Relational Database Machine, Relational Database Machine, RDM 1100, Relational Completeness, Fully Relational, Relational Operations, Database Security, Database Schemas, Relational Database Performance		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This thesis describes the functions of the Database Administrator (DBA) and how they are supported by the benchmarked relational database machine. An examination of the relational query language provided, DBA support services required, the performance issues involved, and the security features employed is presented. The goal of this work is to		

(Continuation of block 20)

develop general guidelines for DBA to follow in implementing and operating an effective, responsive database system.



Approved For	
EX-100-1	<input checked="" type="checkbox"/>
EX-100-2	<input type="checkbox"/>
EX-100-3	<input type="checkbox"/>
Distribution/	
Avail. Class. Sec.	
Dist. For	



Approved for public release, distribution unlimited.

**Benchmarking Relational Database Machines' Capabilities in
Supporting the Database Administrators' Functions and
Responsibilities**

by

**Curtis M. Ryder
Lieutenant Commander, United States Navy
B.S., Florence State University, 1972**

**Submitted in partial fulfillment of the
requirements for the degree of**

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
September 1983**

AUTHOR:

Curtis M. Ryder

APPROVED BY:

David K. Hsia

Thesis Advisor

Paul R. Strawn

Second Reader

David K. Hsia

Chairman, Department of Computer Science

K.T. Manly

Dean of Information and Policy Sciences

ABSTRACT

This thesis describes the functions of the Database Administrator (DBA) and how they are supported by the benchmarked relational database machine. An examination of the relational query language provided, DBA support services required, the performance issues involved, and the security features employed is presented. The goal of this work is to develop general guidelines for DBA to follow in implementing and operating an effective, responsive database system.

TABLE OF CONTENTS

I.	AN INTRODUCTION	10
II.	THE BENCHMARKING ENVIRONMENT	12
	A. THE HOST SYSTEM	12
	1. The Hardware Interface	12
	2. The Software Interface	12
	B. THE BACKEND DATABASE MACHINE	12
	1. A Modular Design	12
	2. The System Configuration	14
III.	THE RELATIONAL QUERY LANGUAGE	16
	A. AN INTRODUCTION TO THE LANGUAGE	16
	B. DATA DEFINITION COMMANDS	17
	1. To Create a Database	18
	2. To Create a Relation	19
	3. To Create an Index	20
	4. To Create a View	20
	5. To Define a Stored Command	21
	6. To Destroy a Database	22
	7. To Destroy an Object	22
	8. To Destroy an Index	22
	C. DATA MANIPULATION COMMANDS	23
	1. To Retrieve Data	23
	2. To Append New Tuples	24
	3. To Replace Attribute Values	24

4.	To Delete Tuples	25
5.	To Aggregate Attribute Values	25
6.	Aggregate Functions	26
7.	String-Manipulation Functions	27
8.	System Supplied Functions	28
D.	EXPRESSING THE RELATIONAL OPERATIONS IN THE QUERY LANGUAGE	29
1.	The Selection Operation	30
2.	The Projection Operation	30
3.	The Join Operation	31
4.	The Division Operation	32
5.	The Union Operation	33
6.	The Intersection Operation	33
7.	The Cartesian-Product Operation	34
8.	The Difference Operation	34
IV.	THE DATABASE ADMINISTRATOR	36
A.	THE DATABASE SYSTEM ENVIRONMENT	37
B.	THE DATABASE DESIGN - THE PHYSICAL AND CONCEPTUAL SCHEMAS	37
1.	Organizational Structure	39
2.	Normalization	40
3.	Database System Architecture	40
4.	Data Sharing and Ownership	41
5.	Recommendations	41
C.	THE DATABASE DESIGN - THE EXTERNAL SCHEMA	42

1.	Permit/Deny Access	43
2.	Create Physical Objects	45
3.	Create Virtual Objects	46
4.	Access Via Stored Commands	46
5.	Recommendations	47
D.	SYSTEM SERVICES	49
1.	System Backup	49
2.	Crash Recovery	51
3.	System Information	52
4.	Translator	57
E.	USER SERVICES	58
1.	Help Facility	58
2.	Stored Commands Provided by DBA	59
F.	SECURITY	60
1.	Security Aspects of 'ALL'	61
2.	System Messages	62
3.	User Identification Numbers	62
4.	Recommendations	62
G.	FINE-TUNING PERFORMANCE	63
1.	Data Reorganization	64
2.	Indices	65
3.	Data Placement	66
V.	EVALUATION OF THE RELATIONAL SYSTEM	67
A.	THE FULLY RELATIONAL SYSTEM	67
1.	The Fully Relational Characteristics ..	67
2.	Four Areas of Deficiency	68

3. The Relational Completeness	70
B. COMPARISON OF TWO QUERY LANGUAGES	70
1. Equal Power	70
2. Differences in the Syntax Structure ...	71
3. Other Differences	74
VI. CONCLUSIONS	76
APPENDIX A (EXAMPLES OF STORED COMMANDS)	79
LIST OF REFERENCES	89
INITIAL DISTRIBUTION LIST	90

ACKNOWLEDGEMENT

The contents of this thesis are the results of the coordinated efforts of numerous individuals. Foremost among these individuals is certainly Ms. Paula Strawser of the Ohio State University and the Naval Postgraduate School. Ms. Strawser's diligence, dedication, and guidance have proven invaluable in both the research prior to and subsequent preparation of this thesis.

Appreciation must also be extended to Dr. David K. Hsiao. Dr. Hsiao's knowledge of database systems provided invaluable insight and made the project a rewarding learning experience.

Likewise, Ms. Doris Mieczko and her staff at the Data Processing Service Center West at Point Mugu, California, have provided much assistance, and they have proven flexible enough to accommodate our sometimes inflexible requirements.

Similarly, Commander Thomas Pigoski of the Naval Security Group Command is offered a special thanks for his continued support in providing the necessary assistance both to keep this project going and to enable the results of this research to be presented at the International Workshop on Database Machines to be held September 1983 in Munich.

Last but not least, gratitude is extended to my wife, Charlotte, who put up with it all.

I. AN INTRODUCTION

Although the application of software database management systems to user requirements is not new there are emerging special-purpose hardware systems which will relieve the host central processing unit (CPU) from the time consuming processes of accessing, updating, and modifying data. Numerous, commercially-available software database management systems for the host computers are currently employed in application areas but there appears to be associated performance degradation in the host machines. These performance issues must be identified and performance measured in order to provide some quantitative comparisons between software systems, general-purpose hardware systems, and special-purpose hardware systems. Historically this information has been collected for general-purpose computers by the use of the instruction mix (Gibson or Flynn) to measure performance in various categories. This measurement of a machine using an instruction as a tool mix is called benchmarking.

The task of benchmarking a database system has not been developed in the literature. Consequently, a research project has been undertaken by the Naval Postgraduate School to develop a set of benchmarking standards which can be employed to obtain a performance index of a particular

database machine/system and further used in a comparative analysis with respect to other database systems/machines.

The initial steps in the benchmark development have been limited to a specific relational database machine. In addition to the measurements of specific database operations, a question of the role and responsibilities of the database administrator (DBA) is posed. With each system benchmarked, there is a need to establish the amount of support provided to DBA. In this case an examination of the facilities provided, query language employed, and amount of additional DBA support required is conducted.

The objective of this thesis is to categorize the duties and responsibilities of DBA and describe how they are supported by the benchmarked system. At the beginning, the system environment is described, followed by a discussion of the query language. An analysis of DBA functions is then made and finally, the fully relational model is examined and a comparison of this particular query language with another well-known language is made.

This thesis is one in a series of four describing the current status of the benchmark development. The other three topics are on generating the synthetic database [Ref. 1], selection and projection [Ref. 2], and join operations [Ref. 3].

II. THE BENCHMARKING ENVIRONMENT

A. THE HOST SYSTEM

The host machine for the benchmark is Univac 1100/42 located at Pacific Missile Test Center, Point Mugu, California. In addition to on-site equipment, a remote terminal is installed at the Naval Postgraduate School.

1. The Hardware Interface

The hardware interface between the host and the database machine is through a Univac 1100/42 I/O channel. This interface channel has a 200-thousand byte/second capacity and the transmission unit is either a byte or a word.

2. The Software Interface

The host software is written by Amperif Corporation of Chatsworth, California. This software consists of the host-driver routines whose primary purpose is to parse the queries and to translate them into the database machine language. Finally, the host handles the communications protocol between the database machine and itself.

B. THE BACKEND DATABASE MACHINE

1. A Modular Design

The database machine which interfaces with the host is an IDM 500 manufactured by Britton-Lee Incorporated of

Los Gatos, California. IDM 500 is being marketed by Amperif Corporation as an RDM 1100. It is a modular, expandable, and microprocessor-based system organized around a central high-speed bus. The separate modules are functionally oriented. The RDM 1100 employs the relational database model which will be discussed in detail in Section V.

The database processor (Z8000-based microprocessor) supervises and manages all system resources. This processor executes most of the software in the system.

The database accelerator is an optional, high-speed processor with an instruction set specifically designed to perform certain relational database functions. The accelerator has a three-stage pipeline which executes instructions at up to 10 MIPS. This processor can initiate disk activity and can process data at disk transfer rates. The accelerator and the RDM 1100 software are so configured that the most frequently occurring database work is performed by the accelerator under the direction of the database processor.

The cache memory (i.e., main memory) of RDM 1100 is composed of 64K-bit chips of dynamic RAM. It may be expanded to a maximum of six megabytes. This cache is utilized for RDM 1100 system code, disk caching, indices, and user commands.

Disk Controller modules may be expanded from one to four. Each controller can manage from one to four disk

drives. The disk controller moves data between the disks and the cache memory, and is designed to work with the accelerator. An optional tape control module supports up to eight tape drives which can be used for direct, disk-to-tape backup, data, and software loading.

RDM 1100 and the host(s) communicate with each other via RDM 1100's host-interface module. This module accepts commands from one or more hosts, and acts on those commands accordingly. Each host-interface module can handle up to eight hosts and a maximum of eight host-interface modules can be made available on RDM 1100. Hence, a maximum of 64 hosts can be accommodated by RDM 1100. In addition to communications handshaking protocols, the interface module performs necessary error checks and causes the host to retransmit any information block in which an error is detected. [Ref. 4]

2. The System Configuration

In the configuration described above (i.e., the connection of the host and the database machine with an I/O channel), the database machine is called a backend database machine. The term, 'backend', is used in this context to refer to a special-purpose machine operating as a peripheral device on one or more host systems. As previously mentioned, the use of the backend machine can significantly reduce the required CPU time for data manipulation by the host. Further advantages are realized through freeing disk

space on the host and the reduction of I/O cycles; thus releasing the CPU to perform other functions necessary for the proper operation of the system and execution of applications programs.

The performance of RDM 1100 is highly dependent on the available hardware configuration. Other performance issues such as indexing and data positioning are dependent on the software developed. The hardware configurations are discussed below and the software issues are discussed in Section IV.

Four test configurations are used during the course of this research. The initial configuration is of one-half megabyte of cache without the accelerator. This configuration will not be marketed by Amperif, but is tested for the purpose of comparison. The next test configuration is of one-megabyte of cache with the accelerator. Following it, a two-megabyte cache with the accelerator is tested. Finally, the accelerator is removed from the configuration. The configuration is tested with only the two-megabyte cache. The standard commercial configuration is with one-megabyte of cache. The accelerator is an optional feature. For specific information on the performance measurements, the reader is directed to [Ref. 2] and [Ref. 3].

III. THE RELATIONAL QUERY LANGUAGE

A. AN INTRODUCTION TO THE LANGUAGE

In addition to the hardware and software to support the host/backend interface, Amperif also provides a language for requesting information or operations on data from the backend database machine. This language is called the Relational Query Language (RQL). The language, being the only interface for the user and database administrator (DBA), is the sole means by which the capabilities and limitations of the backend are known to the user and DBA. Therefore, a discussion about the facilities of RQL will be presented.

This section defines two major command groups available in RQL. The metanotation used in the command syntax consists of the symbols described below.

- | | |
|-----|--|
| () | used as delimiters in RQL |
| [] | used to indicate anything optional inside the square brackets |
| | used to denote a choice of the word either before or after the bar |
| { } | used to specify zero or more occurrences of anything in the curly brackets |
| < > | used as metasympols to denote a construct in |

RQL with the name of the construct between the metasymbols

All other words in RQL are key words and must appear literally [Ref. 5]. In the remaining sections key words are capitalized. In sections explaining the commands, an abbreviated syntax of each command followed by an explanation of the command is provided. This information is taken from [Ref. 5] and [Ref. 6].

B. DATA DEFINITION COMMANDS

In RQL the commands are presented without regard to function. However, in most database books, (e.g., [Ref. 7] and [Ref. 8]), there is a distinction between the data definition language and the data manipulation language. Although this distinction is not made in RQL, it provides a logical division of the majority of commands and facilitates the understanding of the commands. The data definition language consists of those commands which are used for the description of database objects.

Data can be represented in seven different types in RDM 1100. The two-character specifications available are for the compressed (c) and uncompressed (uc) character string with the user providing a maximum length, up to 255 characters. The difference is that the compressed character string is not stored with trailing blanks. Integers can be declared with three different byte sizes namely, 1, 2, or 4.

The byte size of an attribute limits the precision which can be accommodated in the attribute values. Finally, floating-point numbers can also be expressed as compressed (f) or uncompressed (uf). The range provided by these two forms is identical and of 31 significant digits. As in character strings the user must specify the number of significant digits desired. The difference between compressed and uncompressed floating point is the suppression of leading and trailing zeros in the compressed floating point. Compression is a feature designed to reduce the storage requirement in the database. The following declaration is an example of the use of attribute types:

```
name = c25, salary = uf8, age = i1, address = c200.
```

This example establishes four attributes: 'name' whose values can each consist of up to 25 characters, 'salary' whose values are floating-point numbers each of which is of eight significant digits, 'age' whose values are one-byte integers, and 'address' whose values are character strings of up to 200 characters each. Notice 'name' and 'address' are designated as compressed and therefore trailing blanks of their values are not stored.

1. To Create a Database

```
CREATE DATABASE <name> [WITH <options>]
```

This command is used to establish a database which will be referred to by the user-specified name. The two options provided are DISK and DEMAND. DISK allows the specification of one or more disks on which the database will be stored (e.g., DISK = 'sys'). DEMAND specifies the number of 2K-byte blocks to be allocated for the database. If the database grows beyond the allocated blocks, it may be extended with the following command:

EXTEND DATABASE <name> WITH <options>

The options are identical to the options of CREATE DATABASE.

2. To Create a Relation

CREATE RELATION <name> (<attribute name> = <format>
{, <attribute name> = <format>}) [WITH <options>]

The create command is used to establish the schema for a relation. An empty relation is set up in the database when the command is executed with the actual specification of the attributes in parenthesis being given as depicted in the example of data types above. One possible option which may be declared is LOGGING. This option causes every change to the relation to be logged in the database transaction log. This feature is extremely important to maintain the consistency and integrity of the relation when system recovery must be initiated.

3. To Create an Index

```
CREATE [UNIQUE] [NONCLUSTERED | CLUSTERED] INDEX  
[CN] <object name> (<attribute> [, <attribute>])
```

An index on an attribute of a relation provides a direct access to the attribute values in the relation. A unique index on an attribute requires all attribute values to be different. There are two primary differences between clustered and nonclustered indices. A clustered index is nondense (i.e., one entry/block) whereas the nonclustered index is dense (i.e., one entry/tuple). The second difference relates to the storage of data. Although the nonclustered index does not affect the placement of data, the clustered index requires the tuples of the relation to be stored in the order of the attribute values. Consequently, only one clustered index may be created for a relation whereas 250 nonclustered indices may be defined for the same relation. For performance data on operational enhancement provided by indices, see [Ref. 2] and [Ref. 3].

4. To Create a View

```
CREATE VIEW <view name> (<target list>)  
[WHERE <qualification>]
```

The CREATE VIEW command establishes a virtual relation, i.e., there is no storage of tuples associated with the view. A view is a composite relation (without its

own tuples) of attributes from other relations or views. The target list is the list of attributes desired from the other relations or views. Finally, the qualification allows the user to restrict the quantity of data in the view to a particular category and to provide necessary linkages between the relations or views.

5. To Define a Stored Command

```
DEFINE <stored command name>  
    <command> {<command>}  
  
END DEFINE
```

In RQL the DEFINE command provides a mechanism for creating subroutines in the database machine. Stored commands may have parameters or be parameterless. The <command> can be an APPEND, DELETE, REPLACE, RETRIEVE, etc. (to be discussed later). There are two advantages to stored commands. One is that it relieves the operator of retyping a frequently employed command and allows the DBA to provide a simplified method for invoking complex queries. The second and perhaps most important advantage is the performance enhancement. Since the stored command exists in the database with all addresses of cited relations resolved, the communications between the host and the backend machine is reduced to passing an EXEC token and the command name. Examples of stored commands are provided in Appendix A.

6. To Destroy a Database

DESTROY DATABASE <name>

The DESTROY DATABASE command eliminates the entire database by removing all linkages from RDM 1100 and freeing the storage space.

7. To Destroy an Object

DESTROY <object name>

This command eliminates existing relations, established views or stored commands from the database. The space freed by the command is reusable by the database. As indicated previously, views and stored commands depend on existing relations or views. These underlying objects are said to have dependencies. An object which has dependencies cannot be destroyed without first destroying the dependent object. This does not apply to indices, which are automatically destroyed when the relation is destroyed.

8. To Destroy an Index

DESTROY [NONCLUSTERED | CLUSTERED] INDEX [ON]
 <object name> (<attribute name>
 {, <attribute name>})

If an index is unnecessary or the overhead associated with keeping an index is too high, the index may be deleted from a database by the DESTROY INDEX command. In

addition to the object name, the user must also specify the exact attributes of the index for the purpose of avoiding any ambiguity.

C. DATA MANIPULATION COMMANDS

The data manipulation language is that subset of RQL commands which allows the user to access, update, and retrieve the data stored in the database.

1. To Retrieve Data

```
RETRIEVE [UNIQUE] (<target list>) [ORDER [BY] <order
specification> [:A | D]
[, <order specification> [:A | D]]]
[WHERE <qualification>]
```

The RETRIEVE command is the most commonly employed command in RQL. It is the means by which data is extracted from the database and returned to the user. The target list provides the user with the facility to reduce the amount of data by limiting the number of attribute values requested. The format for the target list is:

```
relation_name.attribute_name
[, relation_name.attribute_name].
```

This list of attributes can be from one or more relations. To reduce duplicate information, UNIQUE can be employed. The order specification dictates the order (i.e.,

alphabetic, numeric or alphanumeric order) in which the data is to be sorted. Finally, the qualification allows the user to specify predicates which the data must satisfy and to require linkages between relations. These predicates and linkages reduce the number of tuples retrieved.

2. To Append new Tuples

```
APPEND [TC] <relation name> (<value list>)  
      [WHERE (<qualification>)]
```

The APPEND command allows the user to add tuples to a specific relation. The value list must specify the attribute names and attribute values with an equality sign in between. Unlisted attribute values in the value list are assigned default values (i.e., blanks for characters and zeros for numerals).

3. To Replace Attribute Values

```
REPLACE <relation name> (<value list>)  
      (WHERE <qualification>)
```

REPLACE provides the facilities for updating values stored in the database. Although it can only change one relation at a time, the number of attribute values is not limited. Further, more than one relation can be accessed to calculate what is to be updated. Although a view name may be used in place of the relation name in REPLACE and APPEND commands, the numerous restrictions on the acceptability of

this procedure makes it almost impotent and at best infeasible.

4. To Delete Tuples

```
DELETE <relation_name> [WHERE <qualification>]
```

This command is used to remove one or more tuples from a relation. If a WHERE clause is not specified then all tuples will be deleted.

5. To Aggregate Attribute Values

There are six scalar aggregates supplied in RQL which may be applied to one or more attribute values. These aggregates return a single value, known as the scalar, to the user. The results of MIN and MAX are the smallest and largest attribute values found for the attribute, respectively. SUM and AVG provide the arithmetic total and mean of the respective attribute values. COUNT returns the number of occurrences of the specific attribute value. ANY is used to test for the existence of a specific attribute value. This is accomplished by applying ANY to a condition (e.g., ANY = (relation_name.attribute_name = value)). If the condition is true for at least one attribute value a '1' is returned, '0' otherwise. Any scalar aggregate can have a predicate (qualification) and, since it returns a single value, can be used anywhere a scalar value is permissible in an expression or other predicate. UNIQUE can be used with COUNT, SUM, and AVG to avoid including duplicate entries in

the computed scalar value. For example, COUNT UNIQUE can be used on a personnel database to retrieve the number of different states (assuming birthplace is an attribute) represented by the employees place of birth without regard to the actual number of employees from each state. These scalar aggregates are useful in providing statistics about the database and in isolating tuples whose attribute values are numeric. For example, a query can be composed to provide a list of attribute values such that each value is greater than the average of the values.

6. Aggregate Functions

The term 'function' is misleading when used in this context since the results of applying an aggregate function is a list of scalars. Although this is not the generally accepted concept of a function (returning a single value) in the literature, it will continue to be used in this thesis. Aggregate functions are used in conjunction with the 'group by' (BY) clause. This clause provides a partition of the attribute values. The partitioned values can then be used as arguments of an aggregate function. There can be more than one aggregate function in a query, and aggregate functions may be nested. Additionally, aggregates may appear in both the target list and qualification. An example of the application of an aggregate function can be found in Section V.

The aggregate functions provide the computational power of RQL. Without the functions there would be no easy method of dividing attribute values into sets and performing tests or computations on these value sets. Further, the use of aggregate functions relieves the user from creating numerous temporary relations and from manipulating them individually for the desired result. For example, in a personnel relation with salaries and department numbers as attributes, it may be desirable to compute the average salary of each department. This is easily accomplished by the use of the aggregate function in the target list as follows:

answer = AVG (salary BY dept_no).

If this capability were not available, some other form of partitioning would be required to support the query. One might provide a separate retrieve for each department number, form a temporary relation for the retrieved salary figures, and average on the newly formed relation separately.

7. String-Manipulation Functions

In order to maintain the simple format of a relational system and yet provide the capability to obtain data based on partial or combined attributes, RQL includes three string manipulation functions. The most useful of the three for the user appears to be pattern matching. By using

symbols to represent any number of characters, tuples having a desired internal pattern in a specific attribute can be selected. Again, consider a personnel relation with an attribute of date of birth. The pattern matching function could be used to provide a list of all personnel born in a particular month. Pattern matching applies only to characters and is only used in predicates.

The other two functions are CONCAT and SUBSTRING. These functions can be used with character or binary attributes. CONCAT requires two string arguments, strips all trailing blanks from both strings and concatenates the second string to the first. The SUBSTRING function requires a starting position in the string, a length to define the number of characters desired, and the attribute on which to perform the operation. For an example of SUBSTRING employment, see Appendix A.

8. System Supplied Functions

There are three categories of system supplied functions available in RQL. These provide information about the database and host, cross reference of system assigned identification numbers to associated character strings, and data type conversions.

The first group of functions is parameterless and provide general information about the host and database. For example a user may request the name of the database [DATABASENAME ()], the time or date [GETTIME () or GETDATE

([]), the attached host [HOST ([])], the identity of the DBA [DBA ([])], or who is executing a command [USERID ([])].

The second group of functions is useful in providing information in a meaningful form to the user. There are three self-explanatory commands in this group [REL_ID (relation name), REL_NAME (relation ID), and FIELD_NAME (relation ID, attribute ID)]. These translations are used extensively in Appendix A.

The last group provides the capability to convert expressions (exp) from one data type to another. For example, a user may convert an expression to a 1-, 2- or 4-byte integer [INT1 (exp), INT2 (exp) or INT4 (exp)], a binary number [BIN (exp)], or a floating-point number [FLCAT (length, exp)]. The expression can be any one of the other types listed as well as string and binary coded decimal in their legal forms (e.g., compressed and uncompressed).

D. EXPRESSING THE RELATIONAL OPERATIONS IN THE QUERY LANGUAGE

The power of a relational query language is usually measured by its ability to perform the operations specified in relational algebra or relational calculus. Since the equivalence of the two has been demonstrated [Ref. 8], the relational algebra will be used for comparative purposes without loss of generality. It should be noted that RQL is

probably best characterized as a domain-based relational calculus.

The relational algebra supports four traditional set operations (Union, Intersection, Difference, and Cartesian Product) and four special relational operations (Selection, Projection, Join, and Division). All eight operations will be defined and an example of each in RQL will be provided. In the examples the term, `relation_name`, will be abbreviated to 'rel' and the term, `attribute_name`, to 'attr'.

1. The Selection Operation

The selection operation provides a subset of tuples in a relation which satisfy a given qualification. All attribute values of every tuple satisfying the predicate are included in the subset. RQL provides an ALL keyword which simplifies the selection operation by avoiding the enumeration of every attribute in the target list.

RETRIEVE UNIQUE (rel.ALL) WHERE <qualification>

2. The Projection Operation

Projection is used to reduce the number of attribute values in the tuples which make up the selected subset. In addition to limiting the number of attribute values in a tuple, the projection operation also deletes duplicate tuples from the subset. Deleting duplicates can be enforced by using the optional keyword UNIQUE. Projection in RQL is a function of the target list in the retrieve command. A

qualification may be used to reduce the number of tuples as in selection. To reiterate, selection reduces the number of tuples whereas projection reduces the number of attribute values.

```
RETRIEVE UNIQUE (rel.attr1, rel.attr2, ..., attrn)
WHERE <qualification>
```

3. The Join Operation

The join operation may be performed on any number of relations whose attributes are defined over a common domain. The result of the join is a new, higher-degree relation. Each tuple, in the resultant relation, is formed by concatenating tuples from the source relations whose attribute values satisfy the qualification.

There are different qualifications and therefore different joins. The equi-join is formed over an equality predicate. The inequality join is formed over an inequality predicate with an operator such as <, >, <=, >= or !=. The following is an example of an equi-join; the other joins can be realized by manipulating the target list (natural join) or predicate (inequality join), accordingly.

```
RETRIEVE UNIQUE (rel1.ALL, rel2.ALL)
WHERE rel1.joinattr = rel2.joinattr
```

4. The Division Operation

The division operation is defined for two relations in which the divisor relation has a degree less than the degree of the dividend relation. The resultant relation has a degree equal to the difference of the degrees of the two relations. The division operation is demonstrated using relations rel1 and rel2 and dividing rel1 by rel2 where the degrees of rel1 and rel2 are m and n respectively with $m \geq n$. The resultant relation consists of the first (m-n) attribute values for each tuple in the dividend, rel1, such that every tuple in the divisor, rel2, exists as the last n attribute values of the uniquely determined partial tuples [identical first (m-n) attribute values] in rel1. For example if a relation X has tuples abcd, abef, bccd, and abab, and relation Y has tuples cd and ef then X divided by Y would be the relation containing the tuple ab. The tuple ab would exist in the resultant relation since abcd and abef are in X. However, the tuple bc would not appear since bcef is not in X.

```
RETRIEVE (rel1.attr1, rel1.attr2, ... , rel1.attr(m-n))
        WHERE COUNT (rel2.attr1) =
                CCUNT (rel1.attr1 by rel1.attr1,
                        rel1.attr2, ..., rel1.attr(m-n)
        WHERE rel1.attr(m-n+1) = rel2.attr1
        AND   rel1.attr(m-n+2) = rel2.attr2
```

AND ...

AND rel1.attrm = rel2.attrn)

5. The Union Operation

Union is the traditional set-theoretic definition of union with the additional constraint of requiring the two relations to be union-compatible. Union-compatibility stipulates that the two relations must be of the same degree and the corresponding attribute values must be taken from the same domain (e.g., rel1.attrk and rel2.attrk must be defined over the same domain). The union of two union-compatible relations is the set of all tuples belonging to either relation or both relations. Note that duplicates are not automatically eliminated but a RETRIEVE UNIQUE (union_rel.ALL) can be executed after the following example to display the union.

RETRIEVE INTO union_rel (rel1.all)

WHERE <qualification>

APPEND TO union_rel (attr1 = rel2.attr1,

attr2 = rel2.attr2, ..., attrlast = rel2.attrlast)

WHERE <qualification>

6. The Intersection Operation

Intersection is only defined for union-compatible relations. The resultant relation is comprised of tuples which exist identically in both of the relations.

RETRIEVE UNIQUE (rel1.all)

WHERE rel1.attr1 = rel2.attr1
AND rel1.attr2 = rel2.attr2
AND ...
AND rel1.attrlast = rel2.attrlast

7. The Cartesian-Product Operation

Given two relations, rel1 and rel2, of degree m and n respectively, the Cartesian product is the set of all tuples of degree (m+n) formed by taking the first tuple in rel1 and concatenating to it all tuples (one at a time) in rel2. This process is then repeated for the second tuple in rel1 until all tuples in rel1 have been concatenated with every tuple in rel2.

RETRIEVE UNIQUE (rel1.all, rel2.all)

8. The Difference Operation

The difference of two union-compatible relations is the set of tuples in the first relation but not in the second.

RETRIEVE UNIQUE (rel1.ALL)

WHERE 0 = ANY (rel1.attr1 BY rel1.attr1
WHERE rel1.attr1 = rel2.attr1
AND ...
AND rel1.attrlast = rel2.attrlast)

This query requires each tuple in rel1 to be compared with every tuple in rel2. In the above example, it is assumed that rel1.attr1 is the key for the relation. In the event a relation has a composite key, the rel1.attr1 following the BY can be replaced by a linear list of attributes comprising the key.

IV. THE DATABASE ADMINISTRATOR

The role of DBA is to establish the database and to ensure that the database system is responsive to the user's performance requirements and information needs. Although the discussion of DBA will use RDM 1100 as the target system, the facilities described and DBA support required should be applicable to any relational database management system. In particular, the amount of DBA support required does not depend on a particular system. If the system does not provide certain facilities, DBA will be required to reformat and/or extract the information from the database to satisfy the users information needs. Finally, DBA will be referred to as an individual; however, the functions can be the responsibility of a group of people.

This section will discuss the functions and qualifications of DBA in the areas of database environment, database design, system services, user services, security, and performance enhancement. For each area, a generalized statement concerning DBA functions and qualifications will be provided; then a specific description of the function in the RDM 1100 environment will follow. RDM 1100 feature which supports it.

A. THE DATABASE SYSTEM ENVIRONMENT

A software database management system is designed to support a single database on a general-purpose computer. The advantage of a backend relational database machine is the support which can be provided to multiple hosts and multiple databases. The existence of multiple databases on a single machine creates two levels of management. Level one, the system DBA, is primarily concerned with machine-wide performance and establishing authorizations for the database DBAs. Level two, the database DBAs, is concerned with the operational data in the individual databases. DBA and system DBA should be knowledgeable in the areas outlined above to ensure efficient and reliable database performance.

In RDM 1100 the system DBA has control of a database called the system database. Certain commands such as creating and destroying databases can be issued only from the system database. When a new database is created the individual issuing the CREATE DATABASE command will be DBA for that database. In this thesis DBA will refer to the level-two DBA unless otherwise indicated.

B. THE DATABASE DESIGN - THE PHYSICAL AND CONCEPTUAL SCHEMAS

As alluded to above, DBA has numerous areas of concern. The second area to be addressed is the database design. This topic describes the design of the physical schema and

the conceptual schema. A schema is simply a plan for a particular level of the database. The third level, external schema, will be addressed in Section IV C.

The physical schema, also called the internal schema, is a plan for the actual storage of data on the physical devices available to the database. In RDM 1100 each disk is divided into zones of 180 2K-byte blocks. The first block in each zone is reserved for a directory to the contents of that block. The number of blocks required for a relation is dependent on the number of tuples and the length of the tuples. Since the physical schema is a function of the database system, the major issue from the DBA perspective is whether the system allows the location of data and indices to be explicitly specified.

The conceptual schema is the logical plan normally associated with the entire 'organizational view' and instituted by DBA. As the physical schema is comprised of the actual location and storage structure of the entire database, the conceptual schema includes the names of all relations, indices, and data dictionary entries in the database.

The primary query language subset used to define the conceptual and physical schemas is the data definition language. The mapping between the physical and conceptual schema is performed by the database system. This mapping is built as the objects are made known to the database system.

In RQL the CREATE <object> commands are the primary commands employed to specify the conceptual and physical schemas. The database system will construct a data dictionary for each object. This includes making the object known to the system, reserving appropriate storage, and describing the appearance of the object (e.g., number, size, and type of attributes). In order to design the physical and conceptual schemas, DBA must know the organizational structure and must understand database normalization, the database system architecture, and the concepts of data sharing and ownership.

1. Organizational Structure

Since DBA is responsible for ensuring that the database reflects the 'real world' of the organization it supports, there is ample justification for a good working knowledge of the organization. The objective is to develop a plan which will accurately reflect the organizational requirements without a need to continuously redesign the database. Although it is tempting to limit the application to one functional area like personnel, DBA must be aware of the relationships between the personnel and other entities in the organization. Without a total organizational picture DBA will ultimately be faced with redesign to meet the organization's needs.

2. Normalization

In order to enhance database reliability and reduce redundancy, a solid foundation in relational database design principles is required. One extremely important aspect is the DBA's understanding of normalization. Once a specific normal form is established for a database, DBA must realize the possible implications of deviations from a this normal form and should document the exceptions. Normal forms are not specifically discussed in this thesis and the reader is directed to [Ref. 7] for more information.

The RDM 1100 system, like most existing relational systems, requires only that all relations be in first normal form. This normal form stipulates that each attribute value in a relation must be atomic. That is, the value is not decomposable. Further, there is only a single-value selected from the specific domain for an attribute. Higher normal forms must be enforced by DBA.

3. Database System Architecture

DBA must also understand the architecture of the database system to exploit efficiencies or avoid deficiencies. Since database users do not have static applications and the data stored is also dynamic, DBA must know how to monitor and enhance performance, if possible, when user requirements can no longer be satisfied. requirements.

4. Data Sharing and Ownership

One of the primary reasons for employing a database management system is to share the data among users. This provides a reduction in the storage of redundant data and alleviates the possibility of anomalies associated with redundancy. The concept of sharing data must be tempered with the requirements of user needs and information security. Therefore, a means must be available to provide control over the data and to permit the controlling authority to decide who will have access to the data he controls.

In RDM 1100 the control of data is a function of ownership and access rights. The creator of an object is the owner of that object. Objects which may be owned are databases, relations, views and stored commands. The owner of the object must explicitly permit other users (less DBA) to access the object or portions of an object (e.g., specific relation attribute values). For a more detailed discussion of ownership and access rights, see Section IV C.

5. Recommendations

In database design the first step is to develop a strategy to meet the organizational information requirements. Since the conceptual schema is the comprehensive data description of the organizational information structure, the second step would entail the designing of the conceptual schema. By using this approach,

data independence can be maintained which will prevent the modification of applications programs due to changes in the physical database. Further, the dependencies between the conceptual schema and user requirements can be documented to ensure changes at the conceptual level will not result in changes to the applications programs.

It should be noted that DBA must control the creation of relations and indices in such a manner so that a specific normal form can be maintained. In addition to conforming to the imposed-normal form constraints, each user creating relations to satisfy his own needs must not violate the relations of the database supporting other users. Such violation will certainly contain excessive and redundant information, and undermine the initial database design. Additionally, if individuals sharing a database are permitted to create objects at will, the sharing of data by all users may be subverted and the database could rapidly deteriorate to a user-determined filing system.

C. THE DATABASE DESIGN - THE EXTERNAL SCHEMA

As important as the physical and conceptual schemas are to the implementation of a single database, the establishment of the external schema is critical to the users. In considering divergent user application requirements, the external schema provides the means to define precisely what will satisfy each users information

needs. The external schema of the database is different for each user or group of users. These schemas are composed of subsets of the conceptual schema. The definition of the contents of a particular external schema is normally accomplished through access control of objects existing at the conceptual level. By restricting the relations, attributes, stored commands, and/or views available to a user, a subset of the entire database is defined.

A user's access to the database is determined by the user's access rights. The access rights of a user are authorized by DBA and consequently DBA controls user access to the database. In addition to the verification and matching of host ID and host-user ID to the database system-user ID, these access rights are the only means for access control in the majority of database systems. In RQL the PERMIT and DENY commands on physical objects, virtual objects, and stored commands can be used to establish the various external schemas of the database.

1. Permit/Deny Access

There are two access rights which must be available in a database system to provide a user with the appropriate level of information. These access rights are read and write. Execute privileges can be considered a special case of indirect read/write just as create can be a special case of write.

The two commands in RQL which assign the access rights are PERMIT and DENY. The PERMIT command grants a user a specified access right over an object or command and the DENY command revokes or removes such access rights. DENY is primarily employed to revoke a previously granted PERMIT.

The access rights available in RQL are READ, WRITE, EXECUTE, and CREATE. PERMIT READ provides access to the specified objects (relation, view or named attributes of a relation or view). To modify or add data to existing relations in the database a PERMIT WRITE for the user or group of users on the objects or portions of objects must be explicitly authorized. The keyword ALL can be used to grant read, write, and execute privileges to a user or group of users. Only the owner of the object is authorized to DENY access to the object.

There are two cases of implicit access in RDM 1100. DBA is authorized access to all objects in the database to which he has not been explicitly denied access by the owner. Even if access is denied to DBA by the owner, DBA may still destroy the object by deleting all references to it in the database relations (non-user). Additionally, the owner of any object is permitted access to that object. All other accesses must be authorized by the owner of the object. This is the essence of the access control system.

2. Create Physical Objects

The database management system must provide facilities to create physical objects in the database. Initially, the database must be created with an assigned DBA. Following this, the physical objects in the database must be created. Although an index is not a physical object which may be manipulated by the user, it is discussed in this topic.

In RQL the right to execute the CREATE DATABASE command must be explicitly granted by the system DBA. Once a user has authorization to create a database, the execution of the CREATE DATABASE command makes him DBA of the named database. To add new users to the database, DBA employs the database administrator utilities (DBAU) program. The DBAU NEW_USERS command assigns the host-user ID and host ID and places them in the HOST_USERS relation. The DESTROY DATABASE command can only be executed by DBA, (i.e., the owner of the database).

CREATE <objects> is also controlled through the PERMIT and DENY commands. The permission is similar to CREATE DATABASE in that only DBA for a particular database may authorize the creation of objects. Relations and indices are the physical objects which a user may be authorized to create. Only the owner of a relation is allowed to create an index on the relation using the CREATE INDEX command. However, the DBA must authorize the owner

use of the CREATE INDEX command. Each of the above discussed commands has a counterpart for revocation. Only the owner or DBA may destroy relations and indices.

3. Create Virtual Objects

Once the physical objects are created, it is necessary to create the virtual objects in the conceptual schema which will define the external schema for each user. Views are the virtual objects which a user may be authorized to create.

CREATE VIEW requires the user to have access to the relations over which the view is defined. Only the owner of the view may destroy the view with the DESTROY <object> command.

4. Access Via Stored Commands

Finally, an indirect read/write (execute) access is necessary to allow users to extract information from the database through the use of stored commands. Stored command is an RQL term for a user defined function or procedure. Although this feature may not be available in every database system, it is very useful and powerful when provided. In addition to the efficiency issue of stored commands discussed previously, it is much easier for the user to execute stored commands than to input long queries. In RQL PERMIT EXECUTE allows a specified user or group of users to execute stored commands.

5. Recommendations

There are three methods for providing an external schema to a user or group of users. The first method is through restriction of access on the physical objects in the database. The second method is to define virtual relations which consist only of the necessary subset of data the user is required to access. Finally, the third method entails the extraction of information from the database through the exclusive use of stored commands.

In ROL the major problems with the first two methods are the addition and deletion of data and implementation of ALL. As mentioned in Section III there are too many restrictions on the use of views for updating database. Additional problems can arise using the first method as a result of the system assigning default values to attributes which are not explicitly listed in an APPEND command. For example, an insertion of a tuple with a blank key field (employee number) for a new employee's salary and name would result in a tuple containing the employee's name and salary with a blank key field. A separate insertion containing the employee number and name would result in two tuples in the same relation for a single employee.

The stored command can be executed without granting the user access rights to the relation(s) which are accessed by the command. However, exclusive use of stored commands for information retrieval is not reasonable since

anticipation of every query which a user could possibly require is not possible.

There is a trade-off between access control, performance, and relational perspective. Each of these issues requires the sacrifice of one of the trade-off features. In order to resolve this problem, a combination of the prescribed methods is required. The use of stored commands to input and delete data in a well structured database removes the restrictions on the use of views. Further, stored commands can force the entry of all mandatory attribute values for a tuple through parameter-argument matching which eliminates the duplicate tuple problem described above. Combining stored commands for updating with the use of views to define the external schemas would provide the most logical approach. In order to employ this strategy a major system change is required in the implementation of ALL.

First, it should be obvious that the most logical mechanism for producing an external schema is the view. However, the major problem is the necessity to provide access to the ATTRIBUTE relation to permit the use of ALL with the view name. Therefore, ALL should be implemented such that only the attributes or relations the users are authorized to access are returned. This should not carry an implicit access to the ATTRIBUTE relation. Access to the ATTRIBUTE relation can be restricted by implicit use of

user-id predicates on all queries on data dictionary relations. The performance issue results from the implementation of ALL in the host and the resultant communications between the host and the backend to process a query containing ALL. This performance degradation can be rectified by implementing ALL in the backend relational database machine.

D. SYSTEM SERVICES

The third area is the services provided to DBA by the system. DBA will use these services to facilitate system backup, crash recovery and provide information about the database. The system services establish a nucleus of information and facilities which DBA may be required to augment for his own personal preferences and needs.

1. System Backup

Two areas of system backup must be provided to DBA to ensure proper system functioning. The first area is the necessity of providing a means to record the contents of the database when it is in a consistent state. This is employed most frequently by the system DBA and is addressed further in the next topic on crash recovery.

The second area is the need to return the database to a previous consistent state as a result of aborting a transaction. A transaction is a single command or a series of commands which must be left uncommitted until the final

command has finished. This situation can result from a user decision to abort his transaction prior to completion or the necessity of rolling a transaction back as a result of deadlock. Deadlock occurs in a multiple user system when one user holds a resource (e.g., relations) another user requires and the second user holds a resource the first requires. In this situation, the system is said to be deadlocked since neither user can complete his transaction. To resolve deadlock only one of the transactions must be rolled back. The solution to user aborts is to restore the database to the state it was in prior to the abort.

In RDM 1100 the function of backing up transactions is invisible to DBA. The TRANSACT relation (to be discussed later) is used to maintain the before and after attribute values affected by the transaction for relations created with the logging option. The BATCH relation is used for the other relations. A transaction is by default a single command unless the explicit commands BEGIN before and END TRANSACTION after a group of commands is specified. ABORT TRANSACTION can then be issued after BEGIN and before END to cause rollback. RDM 1100 employs an optimistic concurrency control algorithm which does not prevent deadlock from occurring. The resolution of deadlock is completely invisible to the user and DBA.

2. Crash Recovery

Another facility which must be provided to DBA is the ability to recover from a system malfunction. This is particularly important when the data on the disk has been lost or contaminated. To avoid excessive time delays, periodic copies of the entire database must be made to reduce the amount of updating. The frequency of copying the database is dictated by the number of changes in a period of time and the time demands of the applications programs. The normal method of recovery requires the most recent copy of the database and the transactions which have occurred since the copy was made. Once the copy of the database is loaded, the transactions are rerun to bring the database up to date. Since the chronological list of transactions is the key to recovery, it must be copied from the database on a frequent basis even though the copying of the entire database may be less frequent due to the time required. Of course, some transaction which were in progress or not in a transaction list must be reinitiated by the user.

RDM 1100 provides DUMP DATABASE and LCAD DATABASE commands in the DEAU facility. Additionally, DUMP TRANSACTION is provided to make copies of the transaction log. The command which allows rerunning transactions after a LOAD DATABASE command has been executed is ROLLFORWARD.

3. System Information

The database system employed must provide a data dictionary and statistical information on the database configuration and performance. A data dictionary contains descriptive information about the database. It must include all the various schemas (physical, conceptual, external) and should include cross-reference information such as which programs use what data and synonyms.

In RDM 1100 there are 13 system-supplied database relations which contain descriptive information about the associated database. In addition, there are seven system relations which provide a global description of the database machine.

The system relations provide a catalog of the databases in RDM 1100, a list of disks known to the system, status and types of locks in the system (used for concurrent processing), and the configuration of the communications interface to the attached host(s). Another system relation provides information concerning the activity currently taking place in the database. Two additional relations are used to provide performance data.

Perhaps more important for DBA are the 13 relations associated with each database. Each relation is listed below and a brief description of the type of information contained is provided. The first 11 are used to supply data

dictionary information and the last two provide information related to transaction management.

<u>RELATION NAME</u>	<u>DESCRIPTION</u>
RELATION	A single tuple is provided for each object in the database. This tuple includes, as appropriate, the name of the object, owner, relation identification number, size, location, number of tuples and their length, type of object (user, system, transaction log, file, view or stored command), and the number of attributes.
ATTRIBUTE	A tuple is entered for every attribute in the database. This tuple includes the attribute identification number, data type, maximum length, associated relation ID, and attribute name.
INDICES	Each index has a tuple in the relation. The attributes include the index identification number, relation ID, number of attributes in the index, location, and attribute ID(s).

PROTECT	Contains information associated with the explicit access authorized on objects for users in the database.
QUERY	Contains the stored commands and views.
CROSSREF	Describes the dependencies among relations, indices and stored commands in the database. The dependencies are system defined and not user specified.
USERS	Describes the mappings between user identification numbers, names, and user groups.
HOST_USERS	Defines the mapping between the host ID, host-user ID, and RDM 1100 user ID.
BLOCKALLOC	Catalogs the sector assignment within a zone. Each tuple represents a sector and the assigned object.
DISK_USAGE	Describes database disk allocation.
DESCRIPTIONS	Contains user-specified, textual descriptions of objects and attributes in the database.

BATCH

Contains temporary logging information used by the system for transaction management. This relation provides information on transactions against logged and non-logged relations so they can be canceled if required. The transaction information is held until the transaction is committed.

TRANSACT

Permanent logging information used for crash recovery.

All of the above relations provide the comprehensive picture of the database. Although the information is in the relations, much of it is not in a usable format. For example, only the RELATION relation contains the textual name of a relation. Other relations use the internally assigned relation ID. Further, some of the information is encoded. In order to translate this information into an understandable format DBA must develop stored commands (preferable to ad hoc queries). The number of stored commands will be dependent on the desires of DBA. However, a minimum subset should include commands to list the relations, attributes, indices, attributes in an index, access list associated with an object, description of an object, and dependencies.

The following stored commands are used to yield the minimum subset. TABLES is used to provide a list of objects by type (DBA-supplied parameter to TABLES). For relations, relation name, type, and the number of attributes and tuples in the relation are provided. FIELDS provides the relation name (parameter specified by DBA to FIELDS), attribute name, data type of the attribute (bin, char, int, etc.), and the length of the attribute for every attribute in the relation. ALL_INDICES provides a list of all indices on user relations in the database. The information provided includes the relation name, index identification number, number of attributes in the index, and a narrative description of the type of index. An additional command, INDEX_LIST, is used to provide the same information as ALL_INDICES except a relation/view name is passed as a parameter and only the indices on that relation/view are returned. ATT_IN_INDX1 and ATT_IN_INDX2 are used to list the attributes in an index by name. These commands require two parameters; the index ID and the relation name. The reason for the development of two separate commands is the readability of the output. PROTECTION provides the object name, user name, and type of access authorized for an object which is passed as a parameter. Another command, ACCESS_LIST, is provided to describe an object and the associated access list for a particular object. WHATIS provides a narrative explanation of its parameter from the DESCRIPTION relation. DEPENDS is

used to provide a list of the dependencies on an object. Finally, another useful command is `WHOCREATES` which provides a list of users who have been granted create permission in the database. RQL constructs for the stored commands described above are provided in Appendix A.

4. Translator

Upon implementation of a relational database, it will be necessary to load the data into the system. Since the data exists on some storage device (disk, tape, etc.), there should be a mechanism for presenting the data to the system for immediate loading in a relational format.

In RDM 1100, assuming the data can be collated as a sequence of records on a disk or tape, the 'translator' can then be used to load the database on a relation-by-relation basis. The 'translator' will ask a series of questions to ascertain the incoming data format and establish the relation schema. The following questions must be answered for a relation. The answers are parenthesized.

1. Output directly to the RDM? (y/n)
2. Input file (name)
3. Database (name)
4. Name of table (relation name)
5. Name of 1st field (name of first attribute)
6. Enter input type and length (input file format)
7. Enter output type and length (c12, i1, etc.)

8. Starting position (input file)

(Questions 5 through 8 are repeated for each attribute.)

9. Record length (input file)

E. USER SERVICES

The fourth area is DBA support provided to the users of the relation database system. DBA should provide services/facilities to the users of the database depending on their applications and experience level. A discussion of user services in two general areas will be addressed. These areas are providing a help facility and providing stored commands.

1. Help Facility

As with any interactive system, a help facility is required to preclude time-consuming, trial-and-error corrective procedures. For a relational database system the help facility should include, at a minimum, the syntax and explanation of every language command and an explanation of the stored commands, relations, and views.

In RQL this can be accomplished by creating a help relation with three attributes (object, line number, and text) and defining a stored command which given an object as a parameter will explain its purpose or use. The stored command must contain appropriate predicates in the WHERE clause to ensure the user can only retrieve information from

the help relation about objects which he is authorized access. An example of the help relation and stored command is provided in Appendix A.

2. Stored Commands Provided by DBA

DBA must have an in-depth knowledge of the query language. It is not reasonable to assume that the average user will become proficient in the use of the query language. Both query language complexity and performance issues must be considered. The examples in Sections III and V demonstrate some of the complexities in RQL. DBA will be required to assist the user in the proper formulation of some queries. In addition, the users will look for assistance when confronted with any perceived problem in the database. Since DBA is a database expert, the user will naturally request his assistance.

In addition to applications oriented RQL stored commands, which are not discussed, DBA should provide commands similar to those described earlier in this section for the user. Specifically, DEPENDS, WHATIS, PROTECTION, ATT_IN_INDX1, ATT_IN_INDX2, INDEX_LIST, FIELDS, TABLES, and HELP should be provided. The only difference between the DBA commands and the user's stored commands is the inclusion of the necessary predicates in the WHERE clause to limit the response to data which the user has been granted access. Other minor modifications may also be desired. For example, TABLES could be parameterless and return all relations,

views, and stored commands to which the user has access. PROTECTION could be modified to return only the accesses on objects the user owns.

F. SECURITY

The fifth area for DBA concern is the security of the database. The security of a database system is plagued with the same problems associated with computer security in general. The normal mechanism for security is access control. Since a database system is attached (backend) to a host, the security measures provided by the host are the first level of security afforded the database system. The user ID-password logon procedure employed by general-purpose computers can be used for database systems to provide the same security checks. Additionally, a host ID check in conjunction with the previously mentioned validation can be performed when a backend system is used. Security is also afforded by the backend machine configuration since the database machine is separate from the host and uses its own disks for data storage.

The first security check performed by RDM 1100 is the verification of the host and host-user ID. These IDs are verified each time a request is made from the host to the backend machine. Since the security of the database is closely associated with the security of the host, the use of passwords on the host for identification and verification is

essential. The user ID-password logon procedure is not employed in RDM 1100 but is taken from the host which means there is not an additional ID-password required for the backend machine. In addition to the verification and matching of host ID and host-user ID to the database system-user ID, the access control rights are the only security mechanisms available in RDM 1100.

There are two implicit access rights in RDM 1100. The owner (creator) of any object and DBA are permitted access to that object unless explicitly denied by the owner. All other accesses must be authorized by the owner of the object. This is the essence of the security system. The remainder of this topic will discuss specific security weaknesses in the RDM 1100.

1. Security Aspects of 'ALL'

A crucial aspect for security is the implementation of ALL. ALL is used in a query as a synonym for every attribute of the relation in the target list. As previously discussed, there is not a user ID qualification associated with ALL. Therefore, the translation of ALL to its attribute equivalents is based on the object (relation or view). ALL does not work with a view or a relation unless the user has READ access on the ATRIBUTE relation. However, once this access is authorized, the user can examine the entire conceptual schema which is certainly a violation of security.

2. System Messages

The use of relation.attribute(s) or view.attribute(s) in the target list returns two separate error messages if read access to the object is not permitted. One error message (permission denied on ...) indicates the attribute name is valid but access is not authorized. The other error message (... not found) can be interpreted as the attribute name is non-existent. Although extremely tedious, the error messages can be used in a trail-and-error method to obtain the conceptual schema.

3. User Identification Numbers

Another serious weakness in the security of ROL is the deletion of a user from a database. The easiest method is to delete the user from the HOST_USERS relation which will prohibit him from opening the database. However, if a new user is added to the database from DBAU and the system assigns him the UID which was previously assigned to a deleted user, the new user will inherit all the accesses which were established by DBA and owners for that UID. This is not acceptable since there should not be any implicit authorizations for a new user.

4. Recommendations

The recommendations for correcting the implementation of ALL are discussed in Section IV C5 above. Although not as informative, the return of a single error message for both access denied and relation.attribute not

found would provide less information about the conceptual schema of the database. From a user's perspective it does not appear to be significant whether access is denied or the object is not in the database. The critical issue is to avoid divulgence of the conceptual schema to a user not authorized this information.

The two methods for correcting the user ID problem are the explicit deletion of all access rights in the database (PROTECT, USERS, HCST_USERS) for the old user by DBA, or providing a command in the DBAU to delete a user from a specified database which will explicitly remove all the accesses the user has been granted. The second method is preferable to the first since the system should provide this service to DBA.

G. FINE-TUNING PERFORMANCE

The last area of concern for DBA is the performance enhancement of an existing database. Given that a relational database system has already been selected and the overall performance factors have been established, DBA nevertheless does have a few tools at his disposal which can enhance performance. There are features in the query language implementation which are more efficient than others. For example, a join can run faster depending on which relation is held in cache. One language uses the last relation listed in the query if other factors are equal.

Thus, the order of the relations could be important. Another example is the use of parenthesis to resolve ambiguity in a list of logical predicates. These features are highly implementation-dependent and will not be addressed further. The other three features are data reorganization, indices, and data placement.

In RQL DBA will be required to develop a performance monitoring strategy which may include the periodic execution of stored commands specifically designed to collect performance data.

1. Data Reorganization

As data is added to and deleted from the database, there is an associated fragmentation of relations in physical storage. Even though many database systems provide the capability to reserve extra space for relations, this will result only in a delay of fragmentation. The extent of fragmentation must be monitored and fragmentation eliminated when necessary.

DBA may specify the number of blocks for a database and for a relation in RQL. Additionally, FILLFACTORS can be specified for clustered indices on relations. This FILLFACTOR determines the percentage of each disk block which will be used for the data in the relation when a clustered index is created. When the fragmentation becomes excessive, the clustered index can be destroyed, recreated, and a new FILLFACTOR assigned. This procedure will resort

the data in the blocks available for the relation. A relation will be allowed to grow until it uses all the blocks it is authorized or all blocks in the database are full. Since blocks are not re-used when data is deleted from a relation, this will result in reaching maximum block capacity and fragmentation. DBA can monitor this activity by writing a stored command on the BLOCKS relation. The ability to eliminate fragmentation for a non-indexed relation will depend on the number of free consecutive blocks available in the database. If enough blocks are available, the data can be retrieved into a temporary relation defined over the empty blocks, the original relation destroyed, and the temporary relation renamed. This strategy can also be employed when reclustered does not offer a satisfactory solution to fragmentation.

2. Indices

Indices can enhance the performance of a database for data retrieval. [Ref. 2] and [Ref. 3] have documented the actual enhancement in RDM 1100. Since indices are application-oriented, they are highly desirable for databases where the majority of operations are retrieval of data over large relations or relations which are fairly static can be identified. If numerous update and append transactions are envisioned, then a degradation in performance could result due to the constant updating of the indices. Therefore, DBA should be aware of the size of the

relations and types of operations performed on the relations. For example, if insertion is prevalent then avoidance of indices on the relations which require numerous APPENDs, if possible, may reduce degradation.

3. Data Placement

Hypothetically, the placement of data on disks can enhance performance. For example, if a join between two relations is performed frequently, then placing the relations on separate disks will reduce disk head movement as data is moved into cache. Although this hypothesis has not been verified due to the lack of facilities for placing data in ROL, the data placement strategy should be considered when explicit assignment of physical storage is available. This could be even more significant when processing data on-the-fly is realized, considering the speed discrepancy between reading data and moving disk heads.

V. EVALUATION OF THE RELATIONAL SYSTEM

A. THE FULLY RELATIONAL SYSTEM

1. The Fully Relational Characteristics

The definition of a "fully relational" database management system is given by Chris Date [Ref. 7]. Date suggests that most existing systems are not fully relational. The primary benefit of considering fully relational as a standard and goal for implementation is in the algebraic power of the language and the consistency of system supplied functions. If the system is deficient in any characteristics which Date describes, appropriate action may be taken to provide a semblance of a fully relational system. First, the concept of fully relational is defined; then a comparison of RDM 1100 and RGL to the definition is addressed.

In order for a database to be characterized as fully relational it must support the following:

- a. "relational databases (including the concepts of domain and key and the two integrity rules);"
- b. "a language that is as powerful as the relational algebra (and that would remain so even if all facilities for loops and recursion were to be deleted)." [Ref. 7]

A relational database exhibits the following properties:

- a. Relations are in first normal form.
- b. Associations between relations are explicitly connected through common attributes.
- c. Every value appearing in a given attribute is taken from the domain for that attribute.
- d. Every relation has a unique primary key which distinguishes (identifies) individual tuples.

In addition to the above properties, two integrity rules are required. First, a null value is not permissible as an attribute value of a primary key. Second, if a relation A has an attribute value which is also the primary key of another relation B, then at all times the attribute values in relation A must exist in B. This rule prevents the missing linkages among relations when attribute values are added to relation A or removed from relation B.

2. Four Areas of Deficiency

There are four areas in which RDM 1100 does not meet the requirements for a fully relational system. First, although specification of the schema includes data types for each attribute, no notion of an underlying domain is incorporated. Since attributes are defined by general length and type comparisons of attributes are limited only to similar types (e.g., character with character),

meaningless comparisons are allowed. Without the concepts of sets, enumerated types, and ranges available in higher order languages such as PASCAL or ADA, the support of domains will always be questionable.

Second, the requirement for a unique primary key is not enforced. The uniqueness of an attribute value can be enforced by declaring a unique index on one of the candidate keys. However, this associates an access path with the concept of a key. These are two logically separate issues and as such should be dealt with separately, since the existence of a candidate key does not imply the need for an access path on that attribute.

Third, nulls are not implemented in RDM 1100. However, the default values for integers (zero) and characters (blanks) are provided for unspecified attribute values. Tuple(s) may be entered into a relation without values for the key fields. Even if unique attribute values are enforced through index specification, at least one tuple with the default value in the key attribute will be accepted.

Finally, relations are normally connected through the repetition of some (or all) of the key attribute values in one relation A and in another relation B. However, there is no mechanism to ensure relation B does not contain a value in the connecting attribute which does not exist in relation A.

3. The Relational Completeness

RDM 1100 performs all the relational algebra operations defined in Section III with one exception. This exception deals with the elimination of duplicate tuples in the results after applying certain operators (projection, division, natural join, etc.). For example, although the result relation may appear to satisfy a natural join, it is obvious that duplicates are not a priori eliminated, since the elimination is a function of the associated projection. Additionally, a projection of an attribute in a relation with duplicate entries will return all the values in the attribute without regard to duplicates. A join could be simulated by forming a Cartesian product of the two relations, applying the predicates to the product, extracting the concatenated tuples which satisfy the predicates, and projecting the attributes from the target list.

B. COMPARISON OF TWO QUERY LANGUAGES

This topic provides a comparison of RGL and SQL. The selection of SQL as the comparative language is based on the relative familiarity of a large number of people with the language and its widespread use.

1. Equal Power

The power of the two query languages is practically identical. Both languages are relationally complete which

implies:

- a. Any relation derivable from the database relations using an expression in the relational algebra can be retrieved using the language, and
- b. Any derivable relation can be retrieved using a single statement in the language.

2. Differences in the Syntax Structure

The major difference between SQL and RQL is the syntactic structure. Using the database in Figure 1 from Date, an example of the two query languages will be given.

This example is a query to list the names of all suppliers who do not provide part "P2". As can be seen from inspection of Figure 1 the answer would be one supplier, ADAMS.

```
SQL:          SELECT SNAME
              FROM S
              WHERE 'P2' != ALL
                  (SELECT P_NR
                   FROM SP
```

The query as stated in RQL is:

```
RQL:  RETRIEVE (S.SNAME) WHERE 0 = ANY (SP.P_NR BY
                                      S.SNAME
                                      WHERE SP.P_NR = "P2"
                                      AND S.S_NR = SP.S_NR) GO
```

S

<u>S_NR</u>	<u>SNAME</u>	<u>STATUS</u>	<u>CITY</u>
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

P

<u>P_NR</u>	<u>PNAME</u>	<u>COLOR</u>	<u>WEIGHT</u>	<u>CITY</u>
P1	NUT	RED	12	LONDON
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P4	SCREW	RED	14	LONDON
P5	CAM	BLUE	12	PARIS
P6	COG	RED	19	LONDON

SP

<u>S_NR</u>	<u>P_NR</u>	<u>QTY</u>
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Figure 1. The Supplier-Parts Database

Without regard to implementation the above queries are resolved as follows:

a. In the SQL example the sets of suppliers and the parts they supply is formed by the nested select. Then the outer select will return a supplier's name, if and only if the set of parts supplied by that supplier does not contain "P2".

b. In the RQL example the "by" clause establishes the same set as the inner select of the SQL query. Then the two boolean expressions are evaluated with the "and" conjunction. If no tuples satisfy the conditions for a given supplier, then the value of ANY (tuple) is 0. If ANY is 0, the qualification evaluates to true, and the suppliers name is returned. $S.S_NR = SP.S_NR$ insures that suppliers in the S relation but not in the SP relation are not ignored (i.e., that a supplier who supplies no parts will be included as a tuple in the answer to the query).

The syntactic structure of the example demonstrates the major differences in the two languages. SQL is highly structured, with nested selects. On the other hand, RQL does not permit nesting of retrieves but allows nesting of aggregate functions to perform the same operations. Although it would be purely subjective to favor one method over the other, it appears that the structured approach of SQL may be easier to learn initially. However, once the

aggregate functions of RQL are mastered, the lack of redundancy may be more attractive.

3. Other Differences

RDM 1100 does not implement nulls, but does supply default values (zeros for numeric fields, blanks for character fields). Therefore, the results of the scalar aggregates and aggregate functions (AVG, MIN, MAX, and SUM) are not always predictable. This implies the user must be extremely knowledgeable about the database and use the aggregates with caution (e.g., explicitly exclude zero values from aggregation). In SQL queries can be constructed with "no null" as a qualifier and the tuples with null values in the attribute being aggregated will not be included in the returned value.

SQL uses 'ALL', 'HAVING', 'IN', and others to provide a more set-theoretic description of database manipulation. RQL provides the same capability in the aggregate functions but the concept of set manipulation is not explicit. RQL provides a 'MOD' function and some string manipulation functions which are also available in SQL. The string manipulation functions extend the power of RQL particularly when working with database relations (i.e., non-user defined relations) which have attributes encoded as binary values.

The 'MOD' function is not correctly implemented for negative numbers in RQL or SQL. It returns the modulo class

of the argument as if the argument were a positive integer and 'attaches' the original sign. For example, $-1 \bmod 8 = -(1 \bmod 8) = -1$. To avoid this inconsistency and to correctly implement the mathematical definition, the following nested application of mod is required for modulo 8:

$$\text{mod} (\text{mod} (\text{ARG}, 8) + 8, 8).$$

The actual function implemented appears to be a remainder function which would be consistent since both query languages are implemented in the programming language C. C has a remainder function but not a mod function.

VI. CONCLUSIONS

There are three major areas in which DBA must be knowledgeable in order to ensure the successful management of a database system. These areas are the user services, performance enhancements, and security factors. The specific relational database management system or backend machine employed will dictate the amount of DBA support required in each area.

The user services include the stored commands provided by DBA, the loading of data into the system, the recovery of the database as a result of system malfunction, and a help facility. Although these are not comprehensive and the exact amount of support will be discretionary on the part of DBA, they do form the nucleus for DBA's planning of user support. This support is critical to the acceptance and use of the relational database system by the user community.

The basic tools DBA can use to enhance performance are the implementation of the language features, indices, and data placement. The performance enhancement which can be gained from the query language is only achievable if DBA has a solid understanding of the language and how it is implemented. Certain features of the language will be executed faster than others and since there are numerous ways to form a query to obtain the same information,

knowledge about these characteristics can achieve more rapid responses from the database. Therefore, DBA should review user commands in applications programs and provide guidance to users for the purpose of exploiting the more rapid features. Of course, the specific features will vary between languages.

Indices provide another performance tool in databases where retrieval and joins are the primary operations. Even if these operations are not the most prevalent, indices may still be employed to enhance performance. If the database has a large number of insertion operations, then avoiding the placement of indices on the relations which are changing frequently will not result in serious degradation attributable to the index updating. Additionally, if relations in this type of database which are not subject to frequent insertions but are used in numerous retrieves and joins can be identified; then placement of indices on these relations over the appropriate attributes will enhance the overall performance of the database system.

The ability to explicitly place data in the database should provide a more responsive system. In order to take advantage of data placement DBA must know what relations exist in the system and which ones are joined on a recurring basis.

The security aspects on a relational database system should be a critical issue for DBA. Since a single database

will be used by various users in the organization, there will be data which certain groups of users do not require to perform their functions and more importantly, they should not be allowed to access. Although there is more to security than access control, this appears to be the only mechanism available to DBA to implement a security system in the database. Consequently, access control should be employed to restrict the data available to the users and simultaneously, provide a relational database perspective to each user.

In RDM 1100 there is a trade-off between security, performance, and relational perspective. There were three methods discussed to provide a single external view of the database to a user or group of user. Each of these methods required the sacrifice of one of the trade-off features and in order to resolve this problem, a change in the implementation of ALL is necessary.

The features and issues discussed in this thesis should provide DBA with some guidelines and topics to investigate which will make his database system acceptable and responsive to the users. Although the success or failure of any system cannot be realistically placed on a single individual, it appears DBA will be more responsible than any other person connected with the system if it does not meet the users perceived needs.

APPENDIX A

EXAMPLES OF STORED COMMANDS

ACCESS_LIST

```
DESTROY ACCESS_LIST GO
DEFINE ACCESS_LIST
RETRIEVE (RELATION.NAME, RELATION.TYPE,
        FIELDS = RELATION.ATTCNT, RECORDS = RELATION.TUPS,
        USER = USERS.NAME)
WHERE RELATION.NAME = $0
AND PROTECT.RELID = RELATION.RELID
AND PROTECT.USER = USERS.ID
AND MOD (INT1 (SUBSTRING (1, 1, PROTECT.ATTMAP)), 4) = 1
END DEFINE GO
ASSOCIATE ACCESS_LIST WITH "RETURNS ACCESS LIST FOR AN
                           OBJECT" GO
```

ALL_INDICES

```
DESTROY ISTATUS GO
CREATE ISTATUS (STATUS = I1, DESC = C30) GO
APPEND TO ISTATUS (STATUS = 0,
                  DESC = "NONUNIQ-NONCLUS-NO DEL SILENT")
APPEND TO ISTATUS (STATUS = 1,
```

```

DESC = "UNIQ-NONCLUS-NO DEL SILENT")
APPEND TO ISTATUS (STATUS = 2,
DESC = "NONUNIQ-CLUS-NO DEL SILENT")
APPEND TO ISTATUS (STATUS = 3,
DESC = "UNIQ-CLUS-NO DEL SILENT")
APPEND TO ISTATUS (STATUS = 4,
DESC = "NONUNIQ-NONCLUS-DEL SILENT")
APPEND TO ISTATUS (STATUS = 5,
DESC = "UNIQ-NONCLUS-DEL SILENT")
APPEND TO ISTATUS (STATUS = 6,
DESC = "NONUNIQ-CLUS-DEL SILENT")
APPEND TO ISTATUS (STATUS = 7,
DESC = "UNIQ-CLUS-DEL SILENT")
PERMIT READ OF ISTATUS TO ALL
DENY WRITE OF ISTATUS TO ALL GC
DESTROY ALL_INDICES GO
DEFINE ALL_INDICES
RETRIEVE (REL = REL_NAME (INDICES.RELID), INDICES.INDID,
INDICES.ATTCNT, ISTATUS.DESC)
ORDER BY REL_NAME (INDICES.RELID)
WHERE ISTATUS.STATUS = MOD (MOD (INDICES.STAT, 8) + 8, 8)
AND RELATION.RELID = INDICES.RELID
AND RELATION.TYPE = "U"
END DEFINE GO
ASSOCIATE ALL_INDICES WITH "LIST ALL INDICES ON USER
RELATIONS" GO

```

ATT-IN-INDX1

DESTROY ATT-IN-INDX1 GO

DEFINE ATT-IN-INDX1

RETRIEVE (INDICES.INDID,

ATT1 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(4, 1, INDICES.KEYS))),

ATT2 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(14, 1, INDICES.KEYS))),

ATT3 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(24, 1, INDICES.KEYS))),

ATT4 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(34, 1, INDICES.KEYS))),

ATT5 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(44, 1, INDICES.KEYS))),

ATT6 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(54, 1, INDICES.KEYS))),

ATT7 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(64, 1, INDICES.KEYS))),

ATT8 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(74, 1, INDICES.KEYS))))

WHERE INDICES.INDID = 80

AND INDICES.RELID = REL_ID (81)

END DEFINE GO

ASSOCIATE ATT-IN-INDX1 WITH "LISTS NAMES OF ATTRIBUTES IN

INDEX" GC

ATT-IN-INDX2

DESTROY ATT-IN-INDX2 GO

DEFINE ATT-IN-INDX2

RETRIEVE (INDICES.INDID,

ATT9 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(84, 1, INDICES.KEYS))),

ATT10 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(94, 1, INDICES.KEYS))),

ATT11 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(104, 1, INDICES.KEYS))),

ATT12 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(114, 1, INDICES.KEYS))),

ATT13 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(124, 1, INDICES.KEYS))),

ATT14 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(134, 1, INDICES.KEYS))),

ATT15 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(144, 1, INDICES.KEYS))),

ATT16 = FIELD_NAME (INDICES.RELID, INT1 (SUBSTRING
(154, 1, INDICES.KEYS))))

WHERE INDICES.INDID = 80

AND INDICES.RELID = REL_ID (81)

END DEFINE GO

ASSOCIATE ATT-IN-INDX2 WITH "LISTS NAMES OF ATTRIBUTES IN

INDEX" GO

DEPENDS

```
DESTROY DEPENDS GO
DESTROY OTYPE GO
CREATE OTYPE (TYPE = UC1, DESC = UC15) GO
APPEND TO OTYPE (TYPE = "U", DESC = "USER TABLE      ") GO
APPEND TO OTYPE (TYPE = "S", DESC = "SYSTEM TABLE    ") GO
APPEND TO OTYPE (TYPE = "T", DESC = "TRANSACTION LOG") GO
APPEND TO OTYPE (TYPE = "F", DESC = "FILE           ") GO
APPEND TO OTYPE (TYPE = "V", DESC = "USER VIEW       ") GO
APPEND TO OTYPE (TYPE = "C", DESC = "STORED COMMAND ") GO
DENY WRITE OF OTYPE GO
DENY READ OF OTYPE GO
DEFINE DEPENDS
RETRIEVE (OBJECT = RELATION.NAME, WHICH_IS_A =
          STRING (15, CTYPE.DESC), DEPENDS_ON = $1)
  WHERE CROSSREF.RELID = REL_ID ($1)
    AND CROSSREF.DRELID = RELATION.RELID
    AND OTYPE.TYPE = RELATION.TYPE
END DEFINE GO
ASSOCIATE DEPENDS WITH "LISTS DEPENDENCIES OF THE NAMED
                        OBJECT" GO
```

FIELDS

```
DESTROY FIELDS GO
DESTROY FIELD_EQUIV GO
```

```

CREATE FIELD_EQUIV (NAME = UC4, NUM = 11) GO
APPEND TO FIELD_EQUIV (NAME = "FLT ", NUM = 35) GO
APPEND TO FIELD_EQUIV (NAME = "BIN ", NUM = 45) GO
APPEND TO FIELD_EQUIV (NAME = "CHAR", NUM = 47) GO
APPEND TO FIELD_EQUIV (NAME = "INT ", NUM = 48) GO
APPEND TO FIELD_EQUIV (NAME = "INT ", NUM = 52) GO
APPEND TO FIELD_EQUIV (NAME = "INT ", NUM = 56) GO
DEFINE FIELDS
RETRIEVE (TABLE = RELATION.NAME, FIELD = ATTRIBUTE.NAME,
          TYPE = FIELD_EQUIV.NAME, LEN = ATTRIBUTE.LEN)
WHERE ATTRIBUTE.RELID = RELATION.RELID
AND RELATION.NAME = STABLE.NAME
AND FIELD_EQUIV.NUM = ATTRIBUTE.TYPE
END DEFINE GO
ASSOCIATE FIELDS WITH "RETURNS ALL FIELDS IN THE NAMED
                                RELATION" GO

```

HELP

HELP_REL

OBJECT	LINE_NO	DESCRIPTION
ATT_IN_INDX1	1	THIS IS A STORED COMMAND WHICH HAS
	2	TWO PARAMETERS. THE FIRST PARA-
	3	METER IS THE INDEX ID NO. AND THE
	4	SECOND IS THE RELATION NAME.

5 THESE PARAMETERS MUST BE SEPAR-
6 ATED BY COMMAS. THIS COMMAND
7 PROVIDES THE ATTRIBUTE NAMES OF
8 EACH ATTRIBUTE IN THE INDEX FOR
9 THE GIVEN RELATION OR VIEW. TO
10 EXECUTE THIS COMMAND JUST TYPE
11 IN "HELP" FOLLOWED BY THE OBJECT
12 NAME AND "GO".

DESTROY HELP GO

DEFINE HELP

RETRIEVE (HELP_REL.DESCRPTION)

ORDER BY HELP_REL.LINE_NO : A

WHERE HELP_REL.OBJECT = \$OBJECTNAME

AND PROTECT.RELID = REL_ID (HELP_REL.OBJECT)

AND PROTECT.USER = USERID ()

AND (MOD (INT1 (SUBSTRING (1, 1, PROTECT.ATTMAP)),
4) = 1)

END DEFINE GO

PERMIT EXECUTE OF HELP TO ALL

ASSOCIATE HELP WITH "PROVIDES INFORMATION ABOUT THE OBJECT
PASSED AS A PARAMETER" GO

INDEX_LIST

DESTROY INDEX_LIST GO

DEFINE INDEX_LIST

RETRIEVE (RELATION.NAME, INDICES.INDID, INDICES.ATTCNT,
ISTATUS.DESC)

ORDER BY INDICES.INDID

WHERE INDICES.RELID = RELATION.RELID

AND RELATION.NAME = \$0

AND ISTATUS.STATUS = MOD (MOD (INDICES.STAT, 8)
+ 8, 8)

END DEFINE GO

ASSOCIATE INDEX_LIST WITH "LIST INDICES ON NAMED
RELATION/VIEW" GO

PRCTECTION

DESTROY PTYPE GO

DESTROY ATYPE GO

CREATE PTYPE (ACCESS = I1, DESC = UC15) GO

APPEND TO PTYPE (ACCESS = 1, DESC = "HEAD")

APPEND TO PTYPE (ACCESS = 2, DESC = "WRITE")

APPEND TO PTYPE (ACCESS = 3, DESC = "ALL")

APPEND TO PTYPE (ACCESS = -32, DESC = "EXECUTE")

APPEND TO PTYPE (ACCESS = -53, DESC = "CREATE DATABASE")

APPEND TO PTYPE (ACCESS = -56, DESC = "CREATE")

APPEND TO PTYPE (ACCESS = -58, DESC = "CREATE INDEX") GO

PERMIT READ OF PTYPE GO

DENY WRITE OF PTYPE GO

CREATE ATYPE (ACCESS = I1, DESC = UC8) GO

APPEND TO ATYPE (ACCESS = 1, DESC = "PERMIT")

```

APPEND TO ATYPE (ACCESS = 2, DESC = "DENY      ")
APPEND TO ATYPE (ACCESS = 3, DESC = "BOTH      ") GO
PERMIT READ OF ATYPE GO
DENY WRITE OF ATYPE GO
DESTROY PROTECTION GO
DEFINE PROTECTION
RETRIEVE (ACCESS = CONCAT (ATYPE.DESC, PTYPE.DESC),
          OBJECT = RELATION.NAME, USER = USERS.NAME)
WHERE ATYPE.ACCESS = MOD (INT1 (SUBSTRING (1, 1,
                                           PROTECT.ATTMAP)), 4)

AND PTYPE.ACCESS = PROTECT.ACCESS
AND RELATION.RELID = PROTECT.RELID
AND RELATION.NAME = 80
AND PROTECT.USER = USERS.ID
END DEFINE GO
ASSOCIATE PROTECTION WITH "DISPLAY PROTECTION DATA ABOUT
                           THE NAMED RELATION" GO

```

TABLES

```

DESTROY TABLES GO
DEFINE TABLES GO
RETRIEVE (RELATION.NAME, RELATION.TYPE, FIELDS =
          RELATION.ATTCNT, RECORDS = RELATION.TUPS)
ORDER BY RELATION.NAME : A
WHERE RELATION.TYPE = 80
END DEFINE GO

```

ASSOCIATE TABLES WITH "RETURNS LIST OF RELATIONS, VIEWS OR
STORED CCMMANDS" GC

WHATIS

DESTROY WHATIS GO

DEFINE WHATIS

RETRIEVE (RELATION = REL_NAME (DESCRIPTIONS.RELID),
EXPLANATION = DESCRIPTIONS.TEXT)
WHERE DESCRIPTIONS.RELID = REL_ID (\$0)

END DEFINE GO

ASSOCIATE WHATIS WITH "EXPLAINS WHAT A STORED
COMMAND/RELATION DOES/IS" GO

WHOCREATES

DESTROY WHOCREATES GO

DEFINE WHOCREATES

RETRIEVE (USERS.NAME, PTYPE.DESC)
WHERE PROTECT.USER = USERS.ID
AND (PROTECT.ACCESS = -53 OR PROTECT.ACCESS = -56 OR
PROTECT.ACCESS = -58)
AND PROTECT.ACCESS = PTYPE.ACCESS
AND MOD (INT1 (SUBSTRING (1, 1, PROTECT.ATTMAP)),
4) = 1

END DEFINE GO

ASSOCIATE WHOCREATES WITH "LIST USERS WHO HAVE CREATE
PERMISSION" GO

LIST OF REFERENCES

1. Stone, V. C., Design of Relational Database Benchmarks, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1983.
2. Bogdanowicz, R. A., Benchmarking the Selection and Projection Operations, and Ordering Capabilities of Relational Database Machines, M.S. Thesis, Naval Postgraduate School, Monterey, California, September 1983.
3. Crocker, M. D., Benchmarking the Join Operations of a Relational Database Machine, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1983.
4. Britton Lee, Inc., Intelligent Database Machine Product Description, (no date).
5. Britton Lee, Inc., IDM Software Reference Manual, January 1983.
6. Amperiz Corp., unpublished class notes, (no date).
7. Date, C. J., An Introduction to Database Systems, 3rd ed., Addison-Wesley, 1981.
8. Ullman, J. D., Principles of Database Systems, Computer Science Press, 1980.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4. Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1
5. Dr. D. K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
6. Ms. Paula R. Strawser, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943	1
7. LT Michael D. Crocker, USN PO Box 459 Demopolis, Alabama 36732	1
8. Command Officer Naval Air Station ATTN: Ms. Doris Mieczko, DPSC West (Code 0340) Point Mugu, California 93042	1
9. Commander, Naval Security Group Command ATTN: LCDR Curtis M. Ryder (G 300) 3801 Nebraska Avenue, N. W. Washington, D. C. 20390	1

10. LT Robert A. Bogdanowicz, USN, Code 52 1
1208 Lois Street
Park Ridge, Illinois 60068
11. LCDR Vincent C. Stone, USN 1
1229 San Lions Trail
Martinsville, Virginia 24112
12. Commander, 2
Naval Security Group Command
ATTN: CDR T. M. Pigoski (G 30D)
3801 Nebraska Avenue, N. W.
Washington, D. C. 20390
13. LT Linda E. Widmaier, USN 1
3016 Bromley Court
Woodbridge, Virginia 22192

END

FILMED

02 - 84

DTIC